INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig
Maximilians
Universität
München
LMU

University Timetabling using Constraint Handling Rules

Slim Abdennadher and Michael Marte

In: Journées Francophones de Programmation en Logique et programmation par Contraintes, Nantes, France, 1998
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1998-1, April 1998

University Timetabling using Constraint Handling Rules

Slim Abdennadher and Michael Marte

Computer Science Department, University of Munich
Oettingenstr. 67, 80538 Munich, Germany
E-mail : {abdennad,marte}@informatik.uni-muenchen.de

Abstract *: Timetabling the courses offered at the Computer Science Department of the University of Munich requires the processing of hard and soft constraints. Hard constraints are conditions that must be satisfied, soft constraints however may be violated, but should be satisfied as much as possible. This paper shows how to model our timetabling problem as a partial constraint satisfaction problem and gives a concise finite domain solver implemented with Constraint Handling Rules that, by performing soft constraint propagation, allows for making soft constraints an active part of the problem solving process. Furthermore, we improve efficiency by reusing parts of the previous year's timetable.*

Keywords *: Partial Constraint Satisfaction, Hard and Soft Constraints, Time Tabling, Constraint Handling Rules.*

1 Introduction

The university course timetabling problem consists in scheduling a set of courses within a given number of rooms and time periods. Since timetabling problems differ from one university to another, we concentrated on generating a timetable for our Computer Science Department.

A lot of research has been done on timetabling. The most popular technique is based on graph coloring algorithms [Wer85]. The main disadvantage of this approach is the difficulty of incorporating application-specific constraints into the problem formulation. Other methods include simulated annealing [Abr91] and genetic algorithms [CDM91].

Constraint Logic Programming [JL87, vH89, JM94] (CLP) has become an interesting approach for solving timetabling problems recently. CLP combines the declarativity of logic programming with the efficiency of constraint solving. The timetabling

problem can be elegantly formalized as a constraint satisfaction problem and implemented by means of specialized constraint solving techniques. Since the timetabling problem is usually over-constrained (i.e. not all specified requirements can be satisfied), the representation of the problem requires the processing of soft constraints, that should hold, but might be violated.

Most existing timetabling systems either treat only hard constraints [AB94] or use the following approach to deal with soft constraints [FHS95, HW95]: After the first solution is found, they use a branch-and-bound search that computes the optimum solution of the enumeration procedure, with respect to a given cost function that depends on the set of satisfied soft constraints. Another approach is to adopt constraint solving techniques that have been developed to solve hard constraints [Mey97]. The result of this work is a commercial C++ library providing black-box constraint solvers and search methods for the nurse scheduling problem. Our aim was to implement a similar approach for our timetabling problem using the high-level constraint language Constraint Handling Rules (CHR) [Frü95]. Based on an existing finite domain solver written in CHR we developed a solver which performs hard and soft constraint propagation. This solver takes no more than 20 lines of code. Our system brought down the time necessary for creating a timetable from a few days (by hand) to a few minutes (on a computer).

In this paper we describe the main features of the constraint solver that was used to generate a timetable for the Computer Science Department of the University of Munich. Section 2 introduces our timetabling problem and the constraints that a solution of the problem had to satisfy. Section 3 shows how the problem can be modelled as a partial constraint satisfaction problem. Section 4 gives an overview of the implementation. We conclude with a summary and directions for future work.

2 The Timetabling Problem

Our goal was to find a weekly timetable for the Computer Science Department of the University of Munich. The department offers a five year program for a master degree in computer science consisting of undergraduate studies (two years) and graduate studies (three years).

Our task consisted in generating a timetable from a given set of courses, each associated with its teachers, their personal preferences and the timetable of the previous year. Our system should become part of the overall process of timetabling, which runs as follows.

After collecting wishes of teachers and information on new courses, a first proposal is developed with the timetable of the previous year as a starting point. This is done by using free slots in the timetable left by courses not taking place again for new

courses offered by the same people, whereas wishes of teachers take precedence over the timetable of the previous year. After handing out the proposal to all teachers, evaluations and new wishes are collected.

With the current proposal as a starting point, a next proposal is developed incorporating the responses on the current proposal, again changing as little as possible, and so on. Creating a new timetable is thus a multi-stage, incremental process. Relying on the timetable of the previous year and changing as little as possible by incremental scheduling drastically reduces the amount of work necessary for creating a new timetable and ensures acceptance of the new timetable by keeping the weekly course of events people are accustomed to.

Note that we are only concerned with timetabling, assignment of rooms is done elsewhere. Nevertheless conflicting requirements for space or certain equipment may be a cause for changing the timetable.

The general constraints are due to physical laws, academic reasons and personal preferences of teachers:

- A teacher cannot be at two places at the same time, so do not clash courses of a teacher. There should be at least a one hour break between two courses of a teacher.
- Some teachers prefer certain times or days for teaching.
- Monday afternoon is reserved for professors' meetings: Do not schedule professors' courses for Monday afternoon.
- The department consists of five units, each dedicated to a certain area of research. Most courses are held by members of a single unit while only a few courses are held by members of different units. Courses held by members of a certain unit must not clash with courses held by other members of the same unit.
- An offering typically consists of two lectures and a tutorial per week. There should be a day break between the lectures of an offering. The tutorial should not take place on a day, on which a lecture of the same offering takes place. All courses should be scheduled between 9am and 6pm. No lectures should be scheduled for Friday afternoon. No tutorials should be scheduled for late Friday afternoon.
- Only few of the courses are mandatory for and dedicated to students of a certain term while most courses are optional and open to all students. For each term of the undergraduate studies there is a set of mandatory courses, the attendance of which is highly recommended. Courses of the graduate studies only rely on the knowledge provided by courses of the undergraduate studies. There is

no recommended order of attendance. A term's undergraduate courses must not clash, while different term's undergraduate courses are allowed to clash. Graduate courses should not clash.

First observations made clear that existing timetables do not meet the requirements stated, e.g. courses of a unit or graduate courses clash or an offering's lecture and tutorial are scheduled for the same day. Furthermore, considering the number of graduate courses offered over the years it became clear that there is too little space to schedule all graduate courses without clashes. This is due to the following reason. As mentioned before, undergraduate courses are mandatory and there is a recommended order of attendance. This way it is possible to distinguish first term's students from third term's students and second term's students from fourth term's students, which makes it possible to allow clashing of undergraduate courses of different terms. The graduate courses only rely on the knowledge provided by the undergraduate courses. There is no recommended order of attendance thus making it impossible to distinguish fifth term's students from, e.g., seventh term's students, which makes it necessary to disallow clashing of graduate courses in some way. So we faced two problems:

- The demand for incremental scheduling by basing the new timetable on previous year's timetable and changing as little as possible made it necessary to handle old timetables, which do not meet the requirements stated.
- From a scheduler's point of view the graduate studies lack structure taking freedom and leading to over-constrained timetable specifications.

Tackling the second problem by removing selected no-clash constraints turned out to be laborious and time-consuming and therefore impractical. Classifying graduate courses by contents and expected number of students and allowing clashing of courses of different categories won back some freedom, but it was not possible to identify enough categories in such a way that courses spread evenly over categories, which would have been necessary to prevent conflicts.

It became clear, that we were in need of some kind of weighted constraints able to express weak and strong constraints that are not mandatory.

3 Modeling the Problem as PCSP

A *constraint satisfaction problem* (CSP) is a pair (V, C), where V is a finite set of variables, each associated with a finite domain, and C is a finite set of constraints. A solution of a CSP maps each variable to a value of its domain such that all the constraints are satisfied. A *partial constraint satisfaction problem* (PCSP) [FW92] is

a triple (V, C, ω), where (V, C) is a CSP and ω maps constraints to weights. A requirement's weight expresses the importance of its fulfillment, allowing to distinguish *hard constraints*, which must not be violated, from *soft constraints*, which should not be violated, but may be violated in case this is unavoidable. Hard constraints have an infinite weight. The finite weights of soft constraints allow for the specification of preferences among constraints. A solution of a PCSP maps each variable to a value of its domain such that all hard constraints are satisfied and the total weight of the violated soft constraints is minimal.

Clearly we only need one variable for each course holding the period, i.e. the starting time point, it has been scheduled for. Requirements, wishes and recommendations can be expressed with a small set of specialized constraints:

- *No-clash constraints* demand that a course must not clash with another one.
- *Time constraints* are used to express teachers' preferences and that a course must (not) take place at a certain time or at certain times.
- *Spreading constraints* make sure that there is at least one day (hour) between a course and another one or that two courses are scheduled for different days.
- *Contiguity constraints* make sure that one course will be scheduled directly after another one.

4 Solving the Problem with CHR

Constraint Handling Rules (CHR) [Frü95] is a declarative high-level language extension especially designed for writing constraint solvers. With CHR, one can introduce *user-defined* constraints into a given host language, be it Prolog, Lisp or any other language.

CHR is essentially a committed-choice language consisting of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. There are two kinds of CHR rules: Simplification rules replace constraints by simpler constraints while preserving logical equivalence (e.g. `X > Y, Y > X <=> false`). Propagation rules add new constraints which are logically redundant but may cause further simplification (e.g. `X > Y, Y > Z ==> X > Z`). Repeatedly applying the rules incrementally solves constraints (e.g. `A > B, B > C, C > A` leads to `false`). With multiple heads and propagation rules, CHR provides two features which are essential for non-trivial constraint handling. Due to space limitations, we cannot give a formal account of syntax and semantics of CHR in this paper. An overview on CHR can be found in [Frü95]. Detailed semantics results for CHR are available in [Abd97].

4.1 *The solver*

It is common to use the notation `X::Dom` to state that the value for the variable `X` must be member of the given finite domain `Dom`. More precisely, if `Dom` is a list, then `X` must be a number appearing in `Dom`, and if `Dom` is an interval `A..B`, then `X` must be a number between `A` and `B`.

Constraint solving for finite domains constraints is based on consistency techniques [Mac92]. For example: If the constraint `X::2..6` has been stated already, then stating that `X::4..9` will reduce the domain of `X` by replacing the stated constraints with a new constraint `X::4..6`. The idea is to reduce the domain of variables by removing values that cannot appear in a solution.

This scheme is not sufficient for our needs: Since soft constraints may be violated, the values to be constrained must not be removed from the variable's domain. Moreover, when we have to choose a value for the variable during labeling, we must be able to decide whether a certain value is a good choice or not. Therefore each value must be associated with an assessment. We chose to represent a domain as a list of period-assessment pairs. For example, assume the domain of `X` is `[(3, 0), (4, 1), (5, -1)]`. Then `X` may take one of values `3`, `4` and `5`, whereas `4` is encouraged with assessment `1` and `5` is discouraged with assessment `-1`.

Applying a soft constraint then means to change the assessment of the values to be constrained. For example, assume there is a time constraint with weight `2` stating that `3` should be assigned to `X`. Then we have to increase the assessment for period `3` in the domain of `X` by adding `2` to the current assessment of `3` obtaining the new domain `[(3, 2), (4, 1), (5, -1)]` for `X`. However, applying a hard constraint will still mean to remove values from the variable's domain.

Consequently, our solver is based on three types of constraints:

- `domain(C, S, D)` means that course `C` may only start at some period `S` appearing in `D`, where `D` is a list of period-assessment pairs.
- `in(C, L, W)`: Its meaning depends on the weight `W`. If `W = infinite`, i.e. if the constraint is hard, it means that course `C` must not start at any period *not* appearing in `L`. If `W` is a number, i.e. if the constraint is soft, it means that the assessment for the periods appearing in `L` and not yet removed from the domain of `C` should be *increased* by `W`.
- `notin(C, L, W)`, if hard, means that course `C` must not start at any period appearing in `L`. If it is soft, it means that the assessment for the periods appearing in `L` and not yet removed from the domain of `C` should be *decreased* by `W`.

Before stating the timetable's constraints all variables are initialized by generating a `domain` constraint for each course. We decided for a 24-periods-per-day scheme, the periods being numbered from 0 to 167, 9 denoting 9am on Monday, 33 denoting 9am on Tuesday, and so on. The initial assessment for all periods is 0 indicating that no period is given preference initially.

We now give the core of our solver. An `in` constraint is processed by either pruning the domain or increasing the assessment for the given periods.

```
domain(C, S, D), in(C, L, W) <=>
    (
        W = infinite
    ->  domain_intersection(D, L, D1),
    ;   increase_assessment(W, L, D, D1)
    ),
    domain(C, S, D1).
```

Whenever an `in` constraint arrives, the rule looks for the corresponding `domain` constraint and replaces both by a new `domain` constraint containing a modified domain. For a hard `in` constraint, i.e. in case `W = infinite`, this is the intersection of the domain `D` with the list of periods `L`. For a soft `in` constraint this is `D` with the assessment for the periods appearing in `L` increased by `W` resulting in `D1`. The rule for `notin` is similar:

```
domain(C, S, D), notin(C, L, W) <=>
    (
        W = infinite
    ->  domain_subtraction(D, L, D1),
    ;   decrease_assessment(W, L, D, D1)
    ),
    domain(C, S, D1).
```

Subtracting weights, which are always positive, may result in negative assessments.

Whenever a domain is reduced to the empty list, the timetable's specification is over-constrained. A course cannot be scheduled without violating hard constraints. This is expressed by the following simplification rule:

```
domain(_, _, []) <=> fail.
```

Up to now we only dealt with the low-level constraints of our finite domain PCSP solver. Now we exemplify how to express the application-level constraints in terms of

`in` and `notin` constraints. `no_clash(W, Cs)` means that the courses in the list `Cs` must or should not clash depending on the weight `W`. It gets translated to `notin` constraints. This translation is data-driven: whenever one of the courses, that should or must not clash, has been scheduled for some period, this period gets discouraged or forbidden for the other courses by the following rule.

```
no_clash(W, Cs) <=>
    Cs \= [_],
    select_ground_var(Cs, X, CsRest)
    |
    post_notin_constraints(W, X, CsRest),
    no_clash(W, CsRest).
```

The guard first makes sure that the list of courses `Cs` contains at least two elements. Then it selects a ground variable `X` from `Cs` remembering the other courses in `CsRest`. With no ground variable in `Cs` `select_ground_var` fails. If the guard holds `no_clash(W, Cs)` gets replaced by

- `notin` constraints produced by `post_notin_constraints`, one for each course in `CsRest`, discouraging or forbidding the period `X` and
- a `no_clash` constraint stating that the courses in `CsRest` should or must not clash.

Note that `post_notin_constraints` fails in case `CsRest` contains the value of `X`.

A singleton list of courses means that there is nothing more to do. This case is handled by the following rule.

```
no_clash(_, [_]) <=> true.
```

The translation of the other application-level constraints either follows this scheme or is a one-to-one translation.

The finite domain PCSP solver presented implements arc-consistency for hard binary no-clash, spreading and contiguity constraints. Soft constraint propagation computes assessments for the values not removed by hard constraint propagation. These assessments may be used to inform the search procedure about promising values, thus making soft constraints an active part of the problem solving process.

We implement an n-ary constraint by incrementally spanning a constraint network of binary constraints. For hard no-clash constraints this approach is equivalent to the

one taken in CHIP [vH89] for the `all_different` constraint. We are aware of the bad pruning performance of arc-consistency for hard binary constraints, but its implementation is cheap and it was good enough to solve our problem (compare to [Reg94] for a more complete, but expensive implementation). For soft constraints bad propagation may lead to a bad first solution, violating many of them, but this was not the case for our problem.

4.2 *Labeling*

We employed the *first-fail* strategy to select the course to schedule and a *best-fit* strategy to select a period from the domain of the selected course. First-fail selects one of the most constrained variables, i.e. one of variables with the smallest domain. Best-fit means to choose the period with the best assessment. From an optimistic point of view, this will be the period violating a set of constraints with minimal total weight, but the estimate may be too good due to the incompleteness of the solver for n-ary no-clash constraints. Furthermore, the best assessment does not necessarily violate a minimum number of constraints: a strong personal preference may balance out ten weak no-clash constraints. This approach yielded a good first solution to our problem. It was not necessary to search for a better solution. However, summing up the assessments for the values used in a solution yields a quality criteria which may be used in branch-and-bound to improve this first solution.

4.3 *Reducing Generation Time by Fixing Timetables*

Now, that we have discussed the details of creating a timetable, how do we create a new timetable based on the previous year's timetable with our system? Central to our solution is the notion of *fixing a timetable*. Fixing a timetable involves two steps:

- Adding a hard time constraint for each course that has been scheduled ensuring all courses offered again will be scheduled for the same time.
- Adding a soft time constraint to a teacher's definition for each of the teacher's courses making it likely that periods not used any more will be reused for new courses offered by the teacher.

Thus fixing a timetable accounts for the demand of changing as little as possible. Furthermore it helps reducing the generation time considerably. For example, scheduling 89 courses within 42 time periods taking into account an "almost good" previous timetable took about 2 minutes and 30 seconds, whereas a timetable from scratch took about five minutes.

5 Conclusion and Perspectives

In this paper, we have argued that CHR is a good vehicle for implementing a finite domain PCSP solver, which performs hard and soft constraint propagation. The solver is powerful enough to serve as the core of a university timetabling system.

Our scheduler runs on ECL^{i}PS^e complemented by the CHR library. We rely on the World Wide Web to enable teachers to enter new wishes and offerings into the specification by themselves. The Web frontend is based on HTML, pages are also generated by a Prolog program. It took about two man weeks to write the frontend, only two weeks to write the expert system code and one week to debug it. The solver takes only a few lines of code.

Our prototype has been used to create the summer term's timetable for the Computer Science Department. The good execution time obtained and the very reasonable timetable generated show that with CHR efficient execution and declarative implementation are not incompatible. Our very high-level approach also means that the program can be easily maintained and modified.

One direction for future work is to experiment with variable and value selection heuristics in order to improve the first solution. Furthermore, we intend to implement a finite domain solver for the HCSP scheme proposed by [Mey97] and to use the resulting solver to generate a timetable for a german college. The timetabling problem for colleges is more complex than the one presented here due to the larger number of courses and teachers, and due to the fact that previous timetable cannot be reused.

Acknowledgements

We would like to thank Thom Frühwirth and François Bry for useful comments on a preliminary version of this paper.

References

[AB94] F. Azevedo and P. Barahona. Timetabling in constraint logic programming. In *Proceedings of 2nd World Congress on Expert Systems*, Estoril, Portugal, January 1994.

[Abd97] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP'97*, LNCS 1330. Springer-Verlag, 1997.

[Abr91] D. Abramson. Constructing school timetables using simulated annealing: Sequential and parallel algorithms. *Management Science*, 37, 1:98–113, 1991.

[CDM91] A. Colorni, M. Dorigo, and V. Maniezzo. Genetic algorithms and highly constrained problems: the time-table case. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1*, LNCS 496, pages 55–59, Dortmund, Germany, 1-3 October 1991. Springer-Verlag.

[FHS95] H. Frangouli, V. Harmandas, and P. Stamatopoulos. UTSE: Construction of optimum timetables for university courses — A CLP based approach. In *Proceedings of the Third International Conference on the Practical Applications of Prolog (PAP'95)*, pages 225–243, Paris, France, April 1995.

[Frü95] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer-Verlag, 1995.

[FW92] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21–70, December 1992.

[HW95] M. Henz and J. Wurtz. Using Oz for college time tabling. In *Proceedings of the First International Conference on the Practice and Theory of Automated Timetabling (ICPTAT '95)*, pages 283–296, 1995.

[JL87] J. Jaffar and J. Lassez. Constraint logic programming. In Michael J. O'Donnell, editor, *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, Munich, FRG, January 1987.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 20, 1994.

[Mac92] A. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. Wiley, 1992. Volume 1, second edition.

[Mey97] H. Meyer auf'm Hofe. ConPlan/SIEDAplan: Personnel assignment as a problem of hierarchical constraint satisfaction. In *Proceedings of the 3rd International Conference on the Practical Application of Constraint Technology*, pages 257–272, Blackpool, April 1997.

[Reg94] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[vH89] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1989.

[Wer85] D. De Werra. An introduction to timetabling. *European Journal of Operations Research*, 19:151–162, 1985.